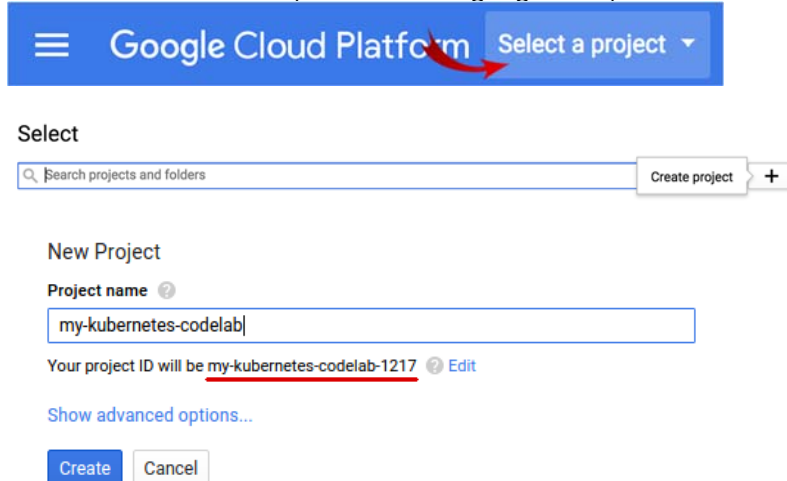**Activity 1 – Classify Text into Categories with the Natural Language API**

In this activity, we will:
- ❏ learn how to classify text through the Google Cloud Natural Language API.
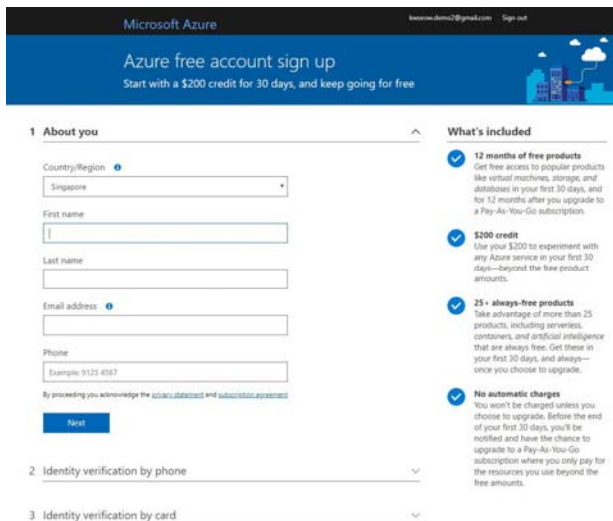- ❏ Use curl command to perform the classification.

1. **Setup and Requirement**
   a) If you don't already have a Google Account (Gmail or Google Apps), you must create one. Sign-in to Google Cloud Platform console (console.cloud.google.com) and create a new project:
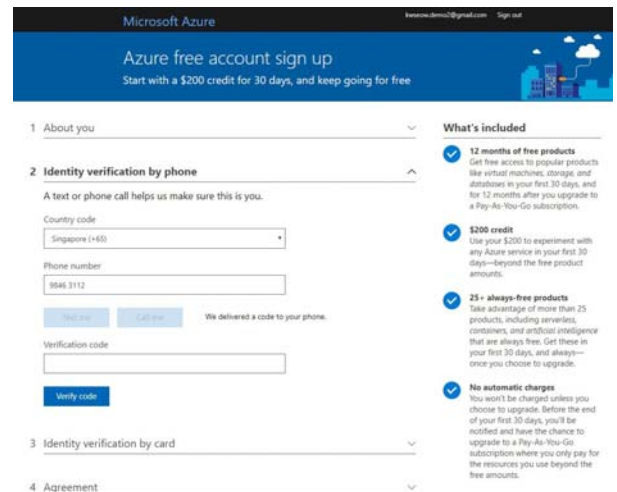

   Remember the project ID, a unique name across all Google Cloud projects (the name above has already been taken and will not work for you, sorry!). It will be referred to later in this exercise as PROJECT_ID.

   b) Next, you'll need to enable billing in the Cloud Console in order to use Google Cloud resources. Running through this exercise shouldn't cost anything, but it could be so if you decide to use more resources or if you leave them running. New users of Google Cloud Platform are eligible for a $300 free trial.

   c) If entered correctly, you be presented with a screen to enter your Azure account information for verification.

   - You will be asked to enter your credit card information for verification too.
   - *You may need to use a different mobile number, credit card and use outlook.com*
   d) Once you are successfully registered, you should see the following.
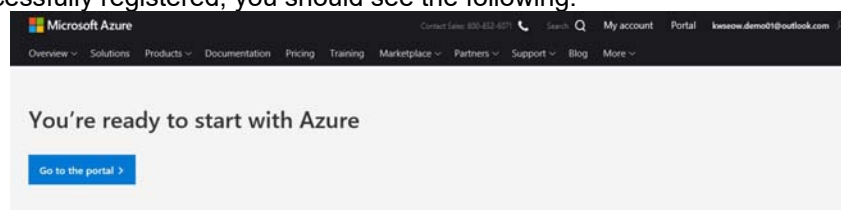
e) Check if you received the $200 credit.

2. Enable the Cloud Natural Language API

a) Click on the menu icon in the top left of the screen.

b) Select **APIs & services** from the drop down and click on Dashboard.
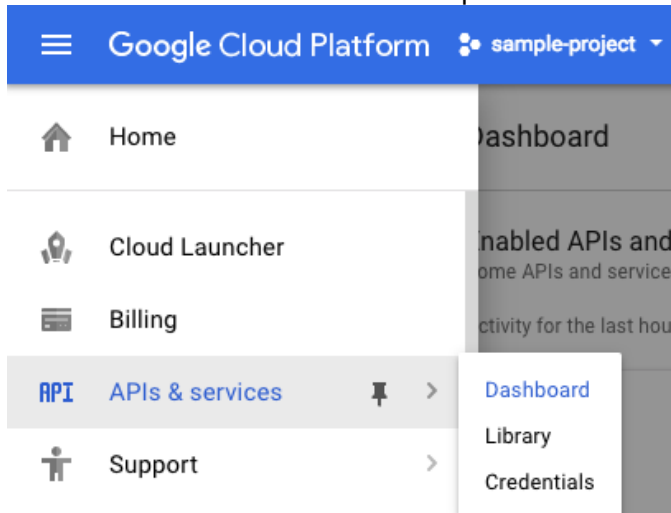
c) Click on **Enable APIs and services**.
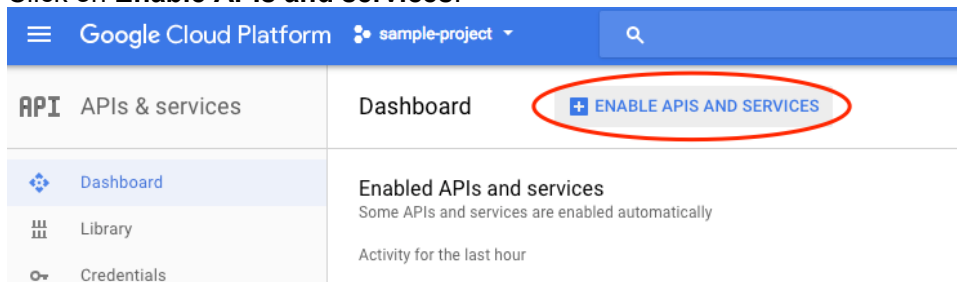
d) Then, search for "language" in the search box. Click on **Google Cloud Natural Language API**:
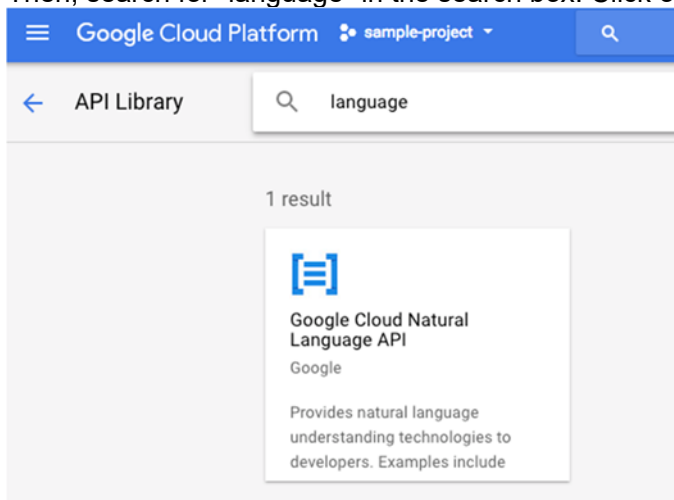
e) Click **Enable** to enable the Cloud Natural Language API:

Wait for a few seconds for it to enable. You will see this once it's enabled:
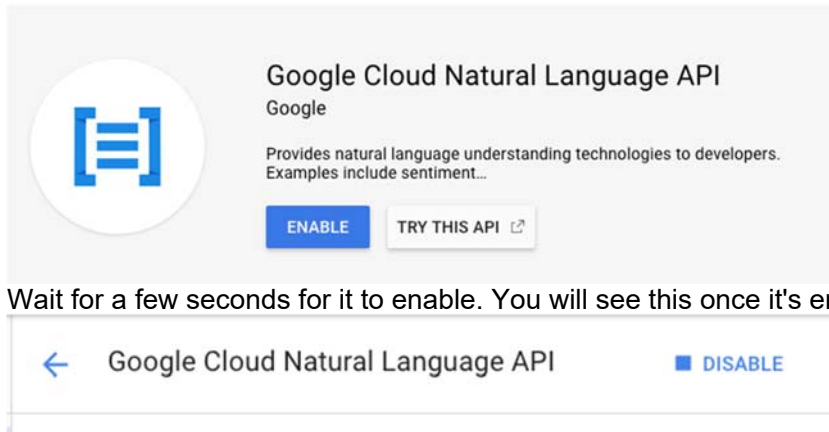
3. Activate Cloud Shell

Google Cloud Shell is a command line environment running in the Cloud. This Debian-based virtual machine is loaded with all the development tools you'll need (gcloud, bq, git and others) and offers a persistent 5GB home directory. We'll use Cloud Shell to create our request to the Natural Language API.

To get started with Cloud Shell, Click on the "Activate Google Cloud Shell" icon in top right hand corner of the header bar

.
A Cloud Shell session opens inside a new frame at the bottom of the console and displays a command-line prompt. Wait until the user@project:~$ prompt appears

4. Create an API Key
   a) Since we'll be using curl to send a request to the Natural Language API, we'll need to generate an API key to pass in our request URL. To create an API key, navigate to the Credentials section of APIs & services in your Cloud console:
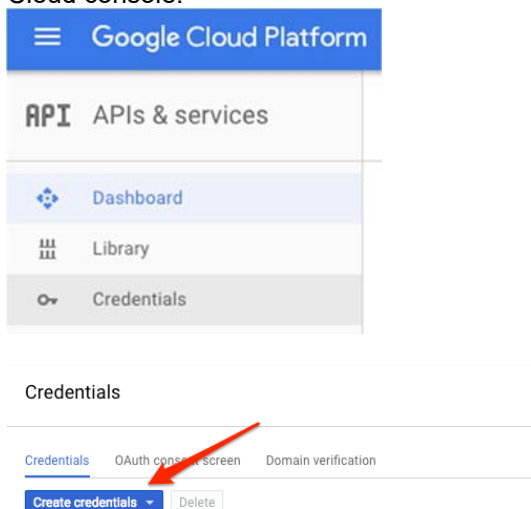
   b) In the drop down menu, select **API key**:
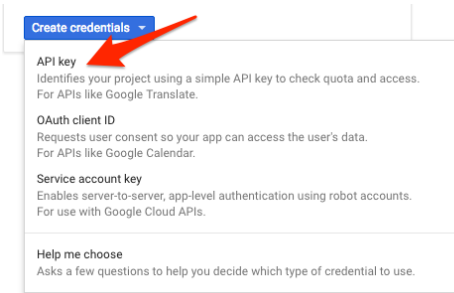
c) Next, copy the key you just generated. You will need this key later in the lab.

d) (Optional) Now that you have an API key, save it to an environment variable to avoid having to insert the value of your API key in each request. You can do this in Cloud Shell. Be sure to replace <your_api_key> with the key you just copied.

```
export API_KEY=<YOUR_API_KEY>
```

On windows, use "set API_KEY=<YOU_API_KEY>"

5. Classify a news article

a) Using the Natural Language API's classifyText method, we can sort our text data into categories with a single API call. This method returns a list of content categories that apply to a text document. These categories range in specificity, from broad categories like /Computers & Electronics to highly specific categories such as /Computers & Electronics/Programming/Java (Programming Language). A full list of the 700+ possible categories can be found here (https://cloud.google.com/natural-language/docs/categories).

b) To start, let's take this headline and description from a New York Times article in the food section:

A Smoky Lobster Salad With a Tapa Twist. This spin on the Spanish pulpo a la gallega skips the octopus, but keeps the sea salt, olive oil, pimentón and boiled potatoes.

In your Cloud Shell environment, create a request.json file with the code below. You can either create the file using one of your preferred command line editors (nano, vim, emacs) or use the built-in Orion editor in Cloud Shell:
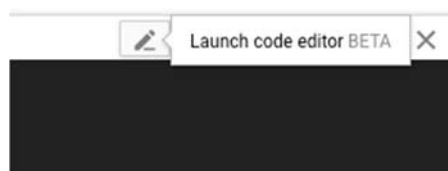
request.json

```json
{
  "document":{
    "type":"PLAIN_TEXT",
    "content":"A Smoky Lobster Salad With a Tapa Twist. This spin on the Spanish pulpo a la gallega skips the octopus, but keeps the sea salt, olive oil, pimentón and boiled potatoes."
  }
}
```

c) Now we can send this text to the NL API's classifyText method with the following curl command:

```
curl "https://language.googleapis.com/v1/documents:classifyText?key=${API_KEY}" \
  -s -X POST -H "Content-Type: application/json" --data-binary @request.json
```

Note: The exercise above runs the request from the Google shell. You can similarly run the classifyText from your local machine.

Let's take a look at the response:

```
{ categories:
```

```
[
  {
    name: '/Food & Drink/Cooking & Recipes',
    confidence: 0.85
  },
  {
    name: '/Food & Drink/Food/Meat & Seafood',
    confidence: 0.63
  }
]
}
```

The API returned 2 categories for this text: /Food & Drink/Cooking & Recipes and /Food & Drink/Food/Meat & Seafood. The text doesn't explicitly mention that this is a recipe or even that it includes seafood, but the API is able to categorize it for us. Classifying a single article is cool, but to really see the power of this feature we should classify lots of text data.

Activity wrap-up:
We learn how to:
❑ Creating a NL API classifyText request and calling the API with curl.

Activity 2 – Entity, Sentiment, and Syntax Analysis with the Natural Language API

In this activity, we will learn:
❑ learn how to extract entities from text, perform sentiment and syntactic analysis through the Google Cloud Natural Language API

1. Setup and Requirements

Like Activity 1, you will need to setup your Google Cloud. Perform Activity 1, step 1 to 4 if you have not done so. Otherwise, you can skip to the next step.

2) Make an Entity Analysis Request

a) The first Natural Language API method we'll use is **analyzeEntities**. With this method, the API can extract entities (like people, places, and events) from text. To try it out the API's entity analysis, we'll use the following sentence:

Joanne Rowling, who writes under the pen names J. K. Rowling and Robert Galbraith, is a British novelist and screenwriter who wrote the Harry Potter fantasy series.

We'll build our request to the Natural Language API in a request.json file. In our case, we will name it entity_analysis.json In your Cloud Shell environment, create the request.json file with the code below. You can either create the file using one of your preferred command line editors (nano, vim, emacs) or use the built-in Orion editor in Cloud Shell:
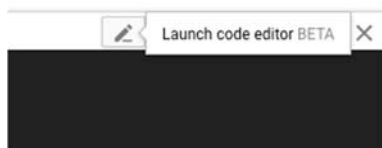
Or you can create the json file on your local machine.

request.json

```
{
  "document":{
    "type":"PLAIN_TEXT",
    "content":"Joanne Rowling, who writes under the pen names J. K. Rowling and Robert Galbraith, is a British novelist and screenwriter who wrote the Harry Potter fantasy series."
  },
  "encodingType":"UTF8"
}
```

In the request, we tell the Natural Language API about the text we'll be sending. Supported type values are *PLAIN_TEXT* or *HTML*. In content, we pass the text to send to the Natural Language API for analysis.

The Natural Language API also supports sending files stored in Cloud Storage for text processing. If we wanted to send a file from Cloud Storage, we would replace content with gcsContentUri and give it a value of our text file's uri in Cloud Storage. encodingType tells the API which type of text encoding to use when processing our text. The API will use this to calculate where specific entities appear in our text.

3) Call the Natural Language API

a) You can now pass your request body, along with the API key environment variable you saved earlier, to the Natural Language API with the following curl command (all in one single command line):

```
curl "https://language.googleapis.com/v1/documents:analyzeEntities?key=${API_KEY}" \
 -s -X POST -H "Content-Type: application/json" --data-binary @entity_analysis.json
```

The beginning of your response should look like the following:

```
{
  "entities": [
    {
      "name": "Robert Galbraith",
```

```
      "type": "PERSON",
      "metadata": {
        "mid": "/m/042xh",
        "wikipedia_url": "https://en.wikipedia.org/wiki/J._K._Rowling"
      },
      "salience": 0.7980405,
      "mentions": [
        {
          "text": {
            "content": "Joanne Rowling",
            "beginOffset": 0
          },
          "type": "PROPER"
        },
        {
          "text": {
            "content": "Rowling",
            "beginOffset": 53
          },
          "type": "PROPER"
        },
        {
          "text": {
            "content": "novelist",
            "beginOffset": 96
          },
          "type": "COMMON"
        },
        {
          "text": {
            "content": "Robert Galbraith",
            "beginOffset": 65
          },
          "type": "PROPER"
        }
      ]
    },

    ...
  ]
}
```

For each entity in the response, we get the entity type, the associated Wikipedia URL if there is one, the salience, and the indices of where this entity appeared in the text. Salience is a number in the [0,1] range that refers to the centrality of the entity to the text as a whole. The Natural Language API can also recognize the same entity mentioned in different ways. Take a look at the mentions list in the response: the API is able to tell that "Joanne Rowling", "Rowling", "novelist" and "Robert Galbriath" all point to the same thing.

4) Sentiment Analysis

a) In addition to extracting entities, the Natural Language API also lets you perform sentiment analysis on a block of text. Our JSON request will include the same parameters as our request above, but this time we'll change the text to include something with a stronger sentiment. Replace your json file with the following, and feel free to replace the content below with your own text:

```json
{
  "document":{
    "type":"PLAIN_TEXT",
    "content":"Harry Potter is the best book. I think everyone should read it."
  },
  "encodingType": "UTF8"
}
```

b) Next we'll send the request to the API's analyzeSentiment endpoint:

```
curl "https://language.googleapis.com/v1/documents:analyzeSentiment?key=${API_KEY}" \
 -s -X POST -H "Content-Type: application/json" --data-binary @request.json
```

Your response should look like this:

```
{
  "documentSentiment": {
    "magnitude": 0.8,
    "score": 0.4
  },
  "language": "en",
  "sentences": [
    {
      "text": {
        "content": "Harry Potter is the best book.",
        "beginOffset": 0
      },
      "sentiment": {
        "magnitude": 0.7,
        "score": 0.7
      }
    },
    {
      "text": {
        "content": "I think everyone should read it.",
        "beginOffset": 31
      },
      "sentiment": {
        "magnitude": 0.1,
        "score": 0.1
      }
    }
  ]
}
```

Notice that we get two types of sentiment values: sentiment for our document as a whole, and sentiment broken down by sentence. The sentiment method returns two values: score and magnitude. score is a number from -1.0 to 1.0 indicating how positive or negative the statement is. magnitude is a number ranging from 0 to infinity that represents the weight of sentiment expressed in the statement, regardless of being positive or negative. Longer blocks of text with heavily weighted statements have higher magnitude values. The score for our first sentence is positive (0.7), whereas the score for the second sentence is neutral (0.1).

5) Analyzing entity sentiment
   a) In addition to providing sentiment details on the entire text document we send the NL API, it can also break down sentiment by the entities in our text. Let's use this sentence as an example:

   I liked the sushi but the service was terrible.

   In this case, getting a sentiment score for the entire sentence as we did above might not be so useful. If this was a restaurant review and there were hundreds of reviews for the same restaurant, we'd want to know exactly which things people liked and didn't like in their reviews. Fortunately the NL API has a method that lets us get the sentiment for each entity in our text, called **analyzeEntitySentiment**. Update your json file with the sentence above to try it out:

```
{
  "document":{
    "type":"PLAIN_TEXT",
    "content":"I liked the sushi but the service was terrible."
  },
  "encodingType": "UTF8"
}
```

b) Then call the analyzeEntitySentiment endpoint with the following curl command:

```
curl "https://language.googleapis.com/v1/documents:analyzeEntitySentiment?key=${API_KEY}" \
 -s -X POST -H "Content-Type: application/json" --data-binary @request.json
```

In the response we get back two entity objects: one for "sushi" and one for "service." Here's the full JSON response:

```
{
  "entities": [
    {
      "name": "sushi",
      "type": "CONSUMER_GOOD",
      "metadata": {},
      "salience": 0.52716845,
      "mentions": [
        {
          "text": {
            "content": "sushi",
            "beginOffset": 12
          },
          "type": "COMMON",
          "sentiment": {
            "magnitude": 0.9,
            "score": 0.9
          }
        }
      ],
      "sentiment": {
        "magnitude": 0.9,
        "score": 0.9
      }
    },
    {
      "name": "service",
      "type": "OTHER",
      "metadata": {},
      "salience": 0.47283158,
      "mentions": [
        {
          "text": {
            "content": "service",
            "beginOffset": 26
          },
          "type": "COMMON",
          "sentiment": {
            "magnitude": 0.9,
            "score": -0.9
          }
        }
      ],
      "sentiment": {
        "magnitude": 0.9,
        "score": -0.9
      }
    }
  ],
  "language": "en"
}
```

We can see that the score returned for "sushi" was 0.9, whereas "service" got a score of -0.9. Cool! You

also may notice that there are two sentiment objects returned for each entity. If either of these terms were mentioned more than once, the API would return a different sentiment score and magnitude for each mention, along with an aggregate sentiment for the entity.

6) Analyzing syntax and parts of speech

a) Using the Natural Language API's method - syntax annotation - we'll dive deeper into the linguistic details of our text. **analyzeSyntax** is a method that provides a full set of details on the semantic and syntactic elements of the text. For each word in the text, the API will tell us the part of speech (noun, verb, adjective, etc.) and how it relates to other words in the sentence (Is it the root verb? A modifier?).

Let's try it out with a simple sentence. Our JSON request will be similar to the ones above, with the addition of a features key. This will tell the API that we'd like to perform syntax annotation. Replace your json file with the following:

```
{
  "document":{
    "type":"PLAIN_TEXT",
    "content": "Hermione often uses her quick wit, deft recall, and encyclopaedic knowledge to help Harry and Ron."
  },
  "encodingType": "UTF8"
}
```

b) Then call the API's analyzeSyntax method:

```
curl "https://language.googleapis.com/v1/documents:analyzeSyntax?key=${API_KEY}" \
 -s -X POST -H "Content-Type: application/json" --data-binary @request.json
```

The response should return an object like the one below for each token in the sentence. Here we'll look at the response for the word "uses":

```
{
  "text": {
    "content": "uses",
    "beginOffset": 15
  },
  "partOfSpeech": {
    "tag": "VERB",
    "aspect": "ASPECT_UNKNOWN",
    "case": "CASE_UNKNOWN",
    "form": "FORM_UNKNOWN",
    "gender": "GENDER_UNKNOWN",
    "mood": "INDICATIVE",
    "number": "SINGULAR",
    "person": "THIRD",
    "proper": "PROPER_UNKNOWN",
    "reciprocity": "RECIPROCITY_UNKNOWN",
    "tense": "PRESENT",
    "voice": "VOICE_UNKNOWN"
  },
  "dependencyEdge": {
    "headTokenIndex": 2,
    "label": "ROOT"
  },
  "lemma": "use"
}
```

Let's break down the response. *partOfSpeech* gives us linguistic details on each word (many are unknown since they don't apply to English or this specific word). *tag* gives the part of speech of this word, in this case a verb. We also get details on the *tense, modality*, and whether the word is singular or plural. *lemma* is the canonical form of the word (for "uses" it's "use"). For example, the words run, runs, ran, and running all have a lemma of run. The lemma value is useful for tracking occurrences of a word in a large piece of text over time.

dependencyEdge includes data that you can use to create a dependency parse tree of the text. This is a diagram showing how words in a sentence relate to each other. A dependency parse tree for the sentence above would look like this:

Hermione often uses her quick wit , deft recall , and encyclopaedic knowledge to help Harry and Ron .

Note: You can create your own dependency parse trees in the browser with the Natural Language demo available here: https://cloud.google.com/natural-language/.

The headTokenIndex in our response above is the index of the token that has an arc pointing at "uses". We can think of each token in the sentence as a word in an array, and the headTokenIndex of 2 for "uses" refers to the word "often," which it is connected to in the tree.

7) Multilingual Natural Language Processing
   a) The Natural Language API also supports languages other than English (full list here). Let's try the following entity request with a sentence in Japanese:

   Modify your json file.

```
{
  "document":{
    "type":"PLAIN_TEXT",
    "content":"日本のグーグルのオフィスは、東京の六本木ヒルズにあります"
  },
  "encodingType": "UTF8"
}
```

   b) Notice that we didn't tell the API which language our text is, it can automatically detect it. Next, we'll send it to the **analyzeEntities** endpoint:
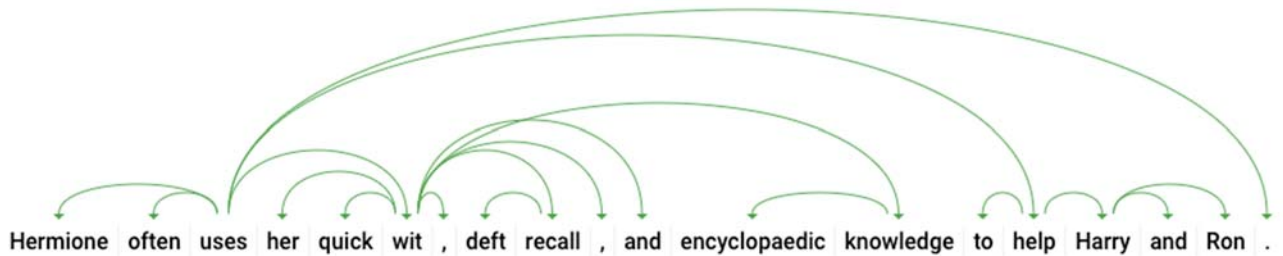
```
curl "https://language.googleapis.com/v1/documents:analyzeEntities?key=${API_KEY}" \
  -s -X POST -H "Content-Type: application/json" --data-binary @request.json
```

   And here are the first two entities in our response:

```
{
  "entities": [
    {
      "name": "日本",
      "type": "LOCATION",
      "metadata": {
        "mid": "/m/03_3d",
        "wikipedia_url": "https://en.wikipedia.org/wiki/Japan"
      },
      "salience": 0.23854347,
      "mentions": [
        {
          "text": {
            "content": "日本",
            "beginOffset": 0
          },
          "type": "PROPER"
        }
      ]
    },
```

```
  {
    "name": "グーグル",
    "type": "ORGANIZATION",
    "metadata": {
      "mid": "/m/045c7b",
      "wikipedia_url": "https://en.wikipedia.org/wiki/Google"
    },
    "salience": 0.21155767,
    "mentions": [
      {
        "text": {
          "content": "グーグル",
          "beginOffset": 9
        },
        "type": "PROPER"
      }
    ]
  },
  ...
 ]
 "language": "ja"
}
```

The API extracts Japan as a location and Google as an organization, along with the Wikipedia pages for each.

Activity wrap-up:
We learn how to
❑ Creating a Natural Language API request and calling the API with curl
❑ Extracting entities and running sentiment analysis on text with the Natural Language API
❑ Performing linguistic analysis on text to create dependency parse trees
❑ Creating a Natural Language API request in Japanese

Activity 3 – Object Recognition with Neural Network

In this activity, we will:
- ❑ load the MNIST dataset in Keras and develop a baseline neural network model for the problem.
- ❑ implement and evaluate a simple Convolutional Neural Network for MNIST.

1. **Setup and Requirement**
   a) A popular demonstration of the capability of deep learning techniques is object recognition in image data. The hello world of object recognition for machine learning and deep learning is the MNIST dataset for handwritten digit recognition.
   b) In this activity, we will discover how to develop a deep learning model to achieve near state-of-the-art performance on the MNIST handwritten digit recognition task in Python using the Keras deep learning library.
   c) Before we start, we will need to install Anaconda
   d) Create virtual environment if desired.
   e) Install a few libraries conda install shown below.

   conda create --name ai4t_py35 --yes python=3.5 tensorflow keras matplotlib
   pip install pillow

2. **Handwritten Digit Recognition Dataset**
   a) The MNIST problem is a dataset developed by Yann LeCun, Corinna Cortes and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem. The dataset was constructed from a number of scanned document datasets available from the National Institute of Standards and Technology (NIST). This is where the name for the dataset comes from, as the Modified NIST or MNIST dataset.

   Images of digits were taken from a variety of scanned documents, normalized in size and centered. This makes it an excellent dataset for evaluating models, allowing the developer to focus on the machine learning with very little data cleaning or preparation required. Each image is a 28 x 28 pixel square (784 pixels total). A standard split of the dataset is used to evaluate and compare models, where 60,000 images are used to train a model and a separate set of 10,000 images are used to test it.

   This is a digit recognition task. As such there are 10 digits (0 to 9) or 10 classes to predict. Excellent results achieve a prediction error of less than 1%. State-of-the-art prediction error of approximately 0.2% can be achieved with large Convolutional Neural Networks.

3. **Loading the MNIST dataset in Keras**
   a) The Keras deep learning library provides a convenience method for loading the MNIST dataset.

```
01   # Plot ad hoc mnist instances
02   from keras.datasets import mnist
03   import matplotlib.pyplot as plt
04   # load (downloaded if needed) the MNIST dataset
05   (X_train, y_train), (X_test, y_test) = mnist.load_data()
06   # plot 4 images as gray scale
07   plt.subplot(221)
08   plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
09   plt.subplot(222)
10   plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
11   plt.subplot(223)
12   plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
13   plt.subplot(224)
14   plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
15   # show the plot
16   plt.show()
```

You can see that downloading and loading the MNIST dataset is as easy as calling the mnist.load_data() function. Running the above example, you should see the image below.
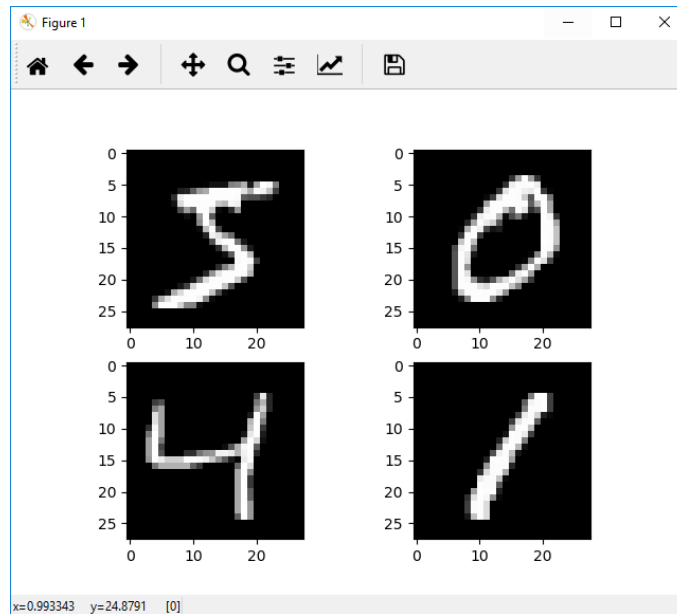
4. Baseline Model with Multilayer Perceptrons

a) You can get good results using a very simple neural network model with a single hidden layer. In this section we will create a simple neural network model that achieves an error rate of approximately 1.73%.

b) Start with loading the necessary libraries and MNIST dataset.

```
01  # Import Classes and Functions
02  import numpy
03  from keras.datasets import mnist
04  from keras.models import Sequential
05  from keras.layers import Dense
06  from keras.layers import Dropout
07  from keras.utils import np_utils
08
09  # It is always a good idea to initialize the random number generator to a constant to ensure
10  # that the results of your script are reproducible.
11  seed = 7
12  numpy.random.seed(seed)
13
14  # load MNIST dataset
15  (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

c) The training dataset is structured as a 3-dimensional array of instance, image width and image height. For a Multilayer Perceptron model we must reduce the images down into a vector of pixels. In this case the 28 x 28 sized images will be 784 pixel input vectors. We can do this transform easily using the reshape() function on the NumPy array. The pixel values are integers, so we cast them to floating point values so that we can normalize them easily in the next step.

```
16  # flatten 28*28 images to a 784 vector for each image
17  num_pixels = X_train.shape[1] * X_train.shape[2]
18  X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
19  X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
```

d) The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using neural network models. Because the scale is well known and well behaved, we can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```
20  # normalize inputs from 0-255 to 0-1
21  X_train = X_train / 255
22  X_test = X_test / 255
```

e) Finally, the output variable is an integer from 0 to 9. This is a multiclass classification problem. As such, it is good practice to use a **one hot encoding** of the class values, transforming the vector of class integers into a binary matrix. We can easily do this using the built-in ***np_utils.to_categorical()*** helper function in Keras.

```
23  # one hot encode outputs
24  y_train = np_utils.to_categorical(y_train)
25  y_test = np_utils.to_categorical(y_test)
26  num_classes = y_test.shape[1]
```

Version 1.0

f) We are now ready to create our simple neural network model. We will define our model in a function. This is handy if you want to extend the example later and try and get a better score.

```
27  # define baseline model
28  def baseline_model():
29      # create model
30      model = Sequential()
31      model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal', activation='relu'))
32      model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))
33      # Compile model
34      model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
35      return model
```

g) The model is a simple neural network with one hidden layer with the same number of neurons as there are inputs (784). A rectifier activation function is used for the neurons in the hidden layer. A softmax activation function is used on the output layer to turn the outputs into probability-like values and allow one class of the 10 to be selected as the model's output prediction. Logarithmic loss is used as the loss function (called categorical_crossentropy in Keras) and the efficient ADAM gradient descent algorithm is used to learn the weights. A summary of the network structure is provided below:
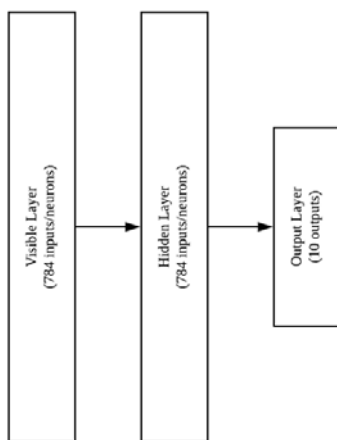
h) We can now fit and evaluate the model. The model is fit over 10 epochs with updates every 200 images. The test data is used as the validation dataset, allowing you to see the skill of the model as it trains. A verbose value of 2 is used to reduce the output to one line for each training epoch. The test dataset is used to evaluate the model and a classification error rate is printed. Finally, the model is saved to a file named mnist_NN.h5.

```
36  # build the model
37  model = baseline_model()
38  # Fit the model
39  model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200, verbose=2)
40  # Final evaluation of the model
41  scores = model.evaluate(X_test, y_test, verbose=0)
42  print("Baseline Error: %.2f%%" % (100-scores[1]*100))
43  print("Metrics(Test loss & Test Accuracy): ")
44  print(scores)
45  # Save the model
46  model.save('mnist_NN.h5')
```

i) Running the example might take a few minutes when run on a CPU. You should see the output below. This simple network dened in very few lines of code achieves a respectable error rate of 1.74%.

```
(ai4t_py35) C:\Users\kwseow\Projects\AI4T\cv>python mnist.py
Using TensorFlow backend.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
2019-07-11 17:29:37.776003: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that
this TensorFlow binary was not compiled to use: AVX AVX2
2019-07-11 17:29:37.784837: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with defaul
t inter op setting: 8. Tune using inter_op_parallelism_threads for best performance.
 - 3s - loss: 0.2782 - acc: 0.9210 - val_loss: 0.1407 - val_acc: 0.9578
Epoch 2/10
 - 3s - loss: 0.1115 - acc: 0.9676 - val_loss: 0.0921 - val_acc: 0.9708
Epoch 3/10
 - 3s - loss: 0.0718 - acc: 0.9796 - val_loss: 0.0789 - val_acc: 0.9763
Epoch 4/10
 - 3s - loss: 0.0505 - acc: 0.9855 - val_loss: 0.0731 - val_acc: 0.9777
Epoch 5/10
 - 3s - loss: 0.0373 - acc: 0.9891 - val_loss: 0.0681 - val_acc: 0.9790
Epoch 6/10
 - 3s - loss: 0.0268 - acc: 0.9928 - val_loss: 0.0626 - val_acc: 0.9802
Epoch 7/10
 - 3s - loss: 0.0208 - acc: 0.9947 - val_loss: 0.0632 - val_acc: 0.9799
Epoch 8/10
 - 3s - loss: 0.0142 - acc: 0.9969 - val_loss: 0.0613 - val_acc: 0.9810
Epoch 9/10
 - 3s - loss: 0.0110 - acc: 0.9977 - val_loss: 0.0585 - val_acc: 0.9816
Epoch 10/10
 - 3s - loss: 0.0081 - acc: 0.9985 - val_loss: 0.0583 - val_acc: 0.9826
Baseline Error: 1.74%
Metrics(Test loss & Test Accuracy):
[0.05825392127795785, 0.9826]
```

j) To test our trained model on our own data, we can create some images and use the model for prediction. For this activity, there are test images that we can use in the data folder. The following script loads the trained model and use the images in the data folder for prediction.

```
01  # Importing the Keras libraries and packages
02  from keras.models import load_model
03  import numpy as np
04  model = load_model('mnist_NN.h5')
05
06  from PIL import Image
07  import numpy as np
08
09  for index in range(10):
10      img = Image.open('data/' + str(index) + '.png').convert("L")
11      img = img.resize((28,28))
12      im2arr = np.array(img)
13      # normalize inputs from 0-255 to 0-1
14      im2arr = im2arr / 255
15      im2arr = im2arr.reshape(1,784)
16      # Predicting the Test set results
17      y_pred = model.predict(im2arr)
18      print(str(index) , "=" , y_pred)
19      print(str(index) , "=" , np.argmax(y_pred))
```

k) Run the above code and you should see the prediction of the images in the data folder. We see that a couple of the digits where predicted wrongly. E.g. 2 and 4.

```
(ai4t_py35) C:\Users\kwseow\Projects\AI4T\cv>python mnist_pred.py
Using TensorFlow backend.
2019-07-11 18:34:52.987493: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that
this TensorFlow binary was not compiled to use: AVX AVX2
2019-07-11 18:34:52.994097: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with defaul
t inter op setting: 8. Tune using inter_op_parallelism_threads for best performance.
0 = [[9.9991119e-01 3.9730967e-08 5.8240144e-05 1.8602566e-05 3.0055372e-09
  9.8536830e-06 2.1294736e-07 7.8387004e-08 2.4013730e-07 1.4470066e-06]]
0 = 0
1 = [[3.8582858e-08 9.9969077e-01 4.7036574e-06 5.0689910e-06 6.9029287e-05
  4.9425048e-06 1.3190304e-06 1.8036216e-04 3.0658743e-05 1.3347055e-05]]
1 = 1
2 = [[1.09987051e-08 5.52849052e-03 1.19092695e-01 1.85047866e-05
  1.35251655e-14 2.36618067e-08 2.91595126e-09 8.75314295e-01
  4.56675880e-05 2.58732740e-07]]
2 = 7
3 = [[4.3297093e-12 8.5711375e-11 3.3283964e-08 9.9990368e-01 4.4932981e-12
  8.5448546e-06 5.1894214e-12 4.1823589e-10 5.4343080e-05 3.3359360e-05]]
3 = 3
4 = [[8.2580262e-11 1.0182876e-07 9.1775945e-07 3.6607453e-08 2.7523620e-07
  2.3214984e-08 6.8676322e-11 9.5542908e-01 5.0928001e-10 4.4569567e-02]]
4 = 7
5 = [[6.46721343e-10 1.30452690e-07 1.96114638e-06 1.79709047e-01
  1.07981506e-07 8.18981349e-01 7.36989250e-07 3.42550788e-09
  1.30661577e-03 6.31970920e-08]]
5 = 5
6 = [[3.8855053e-03 1.2940824e-06 2.0486765e-05 6.6110454e-07 1.7377060e-05
  1.4385127e-02 9.7058558e-01 1.5062104e-05 1.0923691e-02 1.6520862e-04]]
6 = 6
7 = [[1.1729435e-04 1.5797475e-04 1.8849671e-02 7.8909338e-02 5.4159989e-07
  8.6073178e-06 1.5423822e-07 8.9507461e-01 5.2206176e-03 1.6612374e-03]]
7 = 7
8 = [[1.2482708e-06 3.4261677e-07 1.0306154e-02 5.3117273e-04 1.1517158e-08
  4.4348053e-07 4.1134541e-08 1.5913136e-08 9.8915929e-01 1.3048822e-06]]
8 = 8
9 = [[4.5866582e-05 1.7929883e-03 7.2786920e-02 1.5153389e-01 1.0057137e-01
  2.1116806e-03 5.6041731e-06 3.1322300e-02 6.2545884e-01 1.4370531e-02]]
9 = 8
```

5. Simple Convolutional Neural Network for MNIST

a) Now that we have seen how to load the MNIST dataset and train a simple Multilayer Perceptron model on it, it is time to develop a more sophisticated convolutional neural network or CNN model. Keras does provide a lot of capability for creating convolutional neural networks. In this section we will create a simple CNN for MNIST that demonstrates how to use all of the aspects of a modern CNN implementation, including Convolutional layers, Pooling layers and Dropout layers. The first step is to import the classes and functions needed. Load the dataset and prepare the training and testing split.

```
01  # Simple CNN for the MNIST Dataset
02  import numpy
03  from keras.datasets import mnist
04  from keras.models import Sequential
05  from keras.layers import Dense
06  from keras.layers import Dropout
07  from keras.layers import Flatten
08  from keras.layers.convolutional import Conv2D
09  from keras.layers.convolutional import MaxPooling2D
10  from keras.utils import np_utils
11  # fix dimension ordering issue
12  from keras import backend as K
13  K.set_image_dim_ordering('th')
14  # fix random seed for reproducibility
15  seed = 7
16  numpy.random.seed(seed)
17  # load data
18  (X_train, y_train), (X_test, y_test) = mnist.load_data()
19  # reshape to be [samples][channels][width][height]
20  X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
21  X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
22  # normalize inputs from 0-255 to 0-1
23  X_train = X_train / 255
24  X_test = X_test / 255
25  # one hot encode outputs
26  y_train = np_utils.to_categorical(y_train)
27  y_test = np_utils.to_categorical(y_test)
28  num_classes = y_test.shape[1]
```

b) Next we define our neural network model. Convolutional neural networks are more complex than standard Multilayer Perceptrons, so we will start by using a simple structure to begin with that uses all of the elements for state-of-the-art results. Below summarizes the network architecture.

1. The first hidden layer is a convolutional layer called a Conv2D. The layer has 32 feature maps, with the size of 5 x 5 and a rectifier activation function. This is the input layer, expecting images with the structure outline above.

2. Next we define a pooling layer that takes the maximum value called MaxPooling2D. It is configured with a pool size of 2 x 2.
3. The next layer is a regularization layer using dropout called Dropout. It is configured to randomly exclude 20% of neurons in the layer in order to reduce overfitting.
4. Next is a layer that converts the 2D matrix data to a vector called Flatten. It allows the output to be processed by standard fully connected layers.
5. Next a fully connected layer with 128 neurons and rectifier activation function is used.
6. Finally, the output layer has 10 neurons for the 10 classes and a softmax activation function to output probability-like predictions for each class.

```
29  # define a simple CNN model
30  def baseline_model():
31      # create model
32      model = Sequential()
33      model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28), activation='relu'))
34      model.add(MaxPooling2D(pool_size=(2, 2)))
35      model.add(Dropout(0.2))
36      model.add(Flatten())
37      model.add(Dense(128, activation='relu'))
38      model.add(Dense(num_classes, activation='softmax'))
39      # Compile model
40      model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
41      return model
```

As before, the model is trained using logarithmic loss and the ADAM gradient descent algorithm. A depiction of the network structure is provided below.
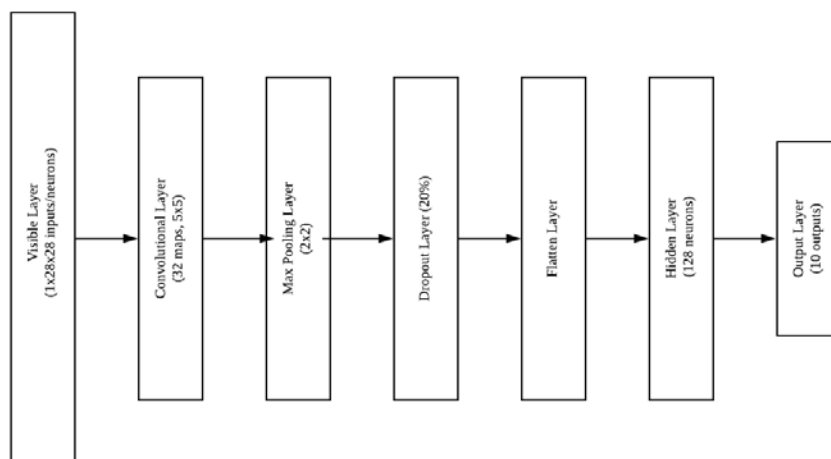
c) We evaluate the model the same way as before with the Multilayer Perceptron. The CNN is fit over 10 epochs with a batch size of 200. Finally, save the model to mnist_CNN.h5.

```
42  # build the model
43  model = baseline_model()
44  # Fit the model
45  model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200)
46  # Final evaluation of the model
47  scores = model.evaluate(X_test, y_test, verbose=0)
48  print("CNN Error: %.2f%%" % (100-scores[1]*100))
49  # Save the model
50  model.save('mnist_CNN.h5')
```

d) Running the example, the accuracy on the training and validation test is printed each epoch and at the end of the classification error rate is printed. The whole training may take 20-30mins on CPU. However, epochs may take about 45 seconds to run on the GPU (e.g. on AWS). You can see that the network achieves an error rate of 1.05, which is better than our simple Multilayer Perceptron model above.

```
(ai4t_py35) C:\Users\kwseow\Projects\AI4T\cv>python mnist_cnn.py
Using TensorFlow backend.
2019-07-11 17:48:25.858775: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that
this TensorFlow binary was not compiled to use: AVX AVX2
2019-07-11 17:48:25.864098: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with defaul
t inter op setting: 8. Tune using inter_op_parallelism_threads for best performance.
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 141s 2ms/step - loss: 0.2241 - acc: 0.9361 - val_loss: 0.0776 - val_ac
c: 0.9756
Epoch 2/10
60000/60000 [==============================] - 146s 2ms/step - loss: 0.0709 - acc: 0.9787 - val_loss: 0.0446 - val_ac
c: 0.9853
Epoch 3/10
60000/60000 [==============================] - 146s 2ms/step - loss: 0.0509 - acc: 0.9842 - val_loss: 0.0432 - val_ac
c: 0.9856
Epoch 4/10
60000/60000 [==============================] - 150s 3ms/step - loss: 0.0391 - acc: 0.9881 - val_loss: 0.0402 - val_ac
c: 0.9873
Epoch 5/10
60000/60000 [==============================] - 147s 2ms/step - loss: 0.0325 - acc: 0.9897 - val_loss: 0.0341 - val_ac
c: 0.9891
Epoch 6/10
60000/60000 [==============================] - 148s 2ms/step - loss: 0.0267 - acc: 0.9916 - val_loss: 0.0326 - val_ac
c: 0.9896
Epoch 7/10
60000/60000 [==============================] - 149s 2ms/step - loss: 0.0218 - acc: 0.9931 - val_loss: 0.0347 - val_ac
c: 0.9884
Epoch 8/10
60000/60000 [==============================] - 147s 2ms/step - loss: 0.0188 - acc: 0.9942 - val_loss: 0.0323 - val_ac
c: 0.9894
Epoch 9/10
60000/60000 [==============================] - 149s 2ms/step - loss: 0.0153 - acc: 0.9951 - val_loss: 0.0308 - val_ac
c: 0.9899
Epoch 10/10
60000/60000 [==============================] - 150s 3ms/step - loss: 0.0138 - acc: 0.9959 - val_loss: 0.0317 - val_ac
c: 0.9895
CNN Error: 1.05%
```

e) To test our trained model on our own data, we can use the following script to load the trained model and use the images in the data folder for prediction.

```
01  # Importing the Keras libraries and packages
02  from keras.models import load_model
03  import numpy as np
04  model = load_model('mnist_CNN.h5')
05
06  from PIL import Image
07  import numpy as np
08
09  for index in range(10):
10      img = Image.open('data/' + str(index) + '.png').convert("L")
11      img = img.resize((28,28))
12      im2arr = np.array(img)
13      # normalize inputs from 0-255 to 0-1
14      im2arr = im2arr / 255
15      im2arr = im2arr.reshape(1,1,28,28)
16      # Predicting the Test set results
17      y_pred = model.predict(im2arr)
18      print(str(index) , "=" , y_pred)
19      print(str(index) , "=" , np.argmax(y_pred))
```

f) Run the above code and you should see the prediction of the images in the data folder. This model is able to recognise every digits correctly!

```
(ai4t_py35) C:\Users\kwseow\Projects\AI4T\cv>python mnist_cnn_pred.py
Using TensorFlow backend.
2019-07-11 18:24:19.000345: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that
this TensorFlow binary was not compiled to use: AVX AVX2
2019-07-11 18:24:19.007535: I tensorflow/core/common_runtime/process_util.cc:69] Creating new thread pool with defaul
t inter op setting: 8. Tune using inter_op_parallelism_threads for best performance.
0 = [[9.9999392e-01 7.8648872e-11 5.7011243e-07 1.0691468e-09 6.0882716e-14
  3.0937881e-11 1.5042194e-06 2.7288330e-10 7.0302474e-07 3.2244902e-06]]
0 = 0
1 = [[2.2214861e-04 9.9340022e-01 1.0397408e-04 9.2971226e-05 2.4428027e-04
  7.8176690e-06 1.7155119e-04 5.5013335e-04 1.3725699e-03 3.8342825e-03]]
1 = 1
2 = [[9.6223713e-13 2.8585969e-08 9.9997151e-01 7.1721961e-06 6.9064618e-18
  2.2408106e-14 2.7199236e-13 4.2489576e-09 2.0939700e-05 3.0522921e-07]]
2 = 2
3 = [[1.6863967e-14 5.1837190e-14 1.4711668e-08 9.9994576e-01 1.5128292e-15
  1.6638452e-08 2.7569355e-14 5.1269371e-09 3.8710546e-06 5.0436603e-05]]
3 = 3
4 = [[1.7027265e-08 3.7747220e-06 1.6308868e-04 3.9450279e-06 9.9721032e-01
  5.6426874e-08 1.8417687e-07 5.1830311e-06 3.5563372e-07 2.6130460e-03]]
4 = 4
5 = [[2.3690839e-13 1.1439558e-11 7.7696606e-13 3.4067690e-02 4.5750331e-11
  9.6589035e-01 9.1382077e-08 1.6903267e-09 4.1444600e-05 3.2810615e-07]]
5 = 5
6 = [[2.7152575e-05 1.1112229e-10 2.1888973e-06 4.3354407e-07 1.6626235e-09
  2.9222216e-03 9.7832799e-01 4.2897844e-07 1.8462211e-02 2.5733115e-04]]
6 = 6
7 = [[3.3346062e-08 2.4278052e-07 4.0536365e-03 3.3581124e-05 3.5849639e-11
  6.1303115e-11 7.3945028e-13 9.9590343e-01 5.3172644e-06 3.7842697e-06]]
7 = 7
8 = [[7.4013906e-09 5.7200186e-12 3.1746100e-03 2.1196485e-07 2.0396174e-12
  2.7657685e-11 7.8205542e-10 4.4914010e-11 9.9682522e-01 5.7387801e-09]]
8 = 8
9 = [[9.8510022e-10 4.6417401e-11 5.0385671e-07 1.3220594e-02 4.1195240e-06
  1.0919827e-06 2.2336498e-10 3.5824171e-06 8.9584777e-05 9.8668057e-01]]
9 = 9
```

As an exercise, you may want to try out a larger CNN, e.g.

```
01  def larger_model():
02    # create model
03    model = Sequential()
04    model.add(Conv2D(30, (5, 5), input_shape=(1, 28, 28), activation='relu'))
05    model.add(MaxPooling2D(pool_size=(2, 2)))
06    model.add(Conv2D(15, (3, 3), activation='relu'))
07    model.add(MaxPooling2D(pool_size=(2, 2)))
08    model.add(Dropout(0.2))
09    model.add(Flatten())
10    model.add(Dense(128, activation='relu'))
11    model.add(Dense(50, activation='relu'))
12    model.add(Dense(num_classes, activation='softmax'))
13    # Compile model
14    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
15    return model
```

Activity wrap-up:
We learn how to
❑ load the MNIST dataset in Keras and develop a baseline neural network model for the problem.
❑ implement and evaluate a simple Convolutional Neural Network for MNIST.